OpenMP를 이용한 Recursive Digital Filter의 Multi-core System 구현

이동환, 성원용
서울대학교 전기공학부
e-mail : ldh@dsp.snu.ac.kr, wysung@snu.ac.kr

OpenMP based Implementation of Recursive Digital Filters in Multi-core Systems

Donghwan Lee and Wonyong Sung
School of Electrical Engineering
Seoul National University

Abstract

Direct parallel computation of recursive digital filtering equations is not allowed due to the dependency problem caused by feedback. An efficient parallel block processing program for recursive digital filtering equations is developed using OpenMP API for multi-core systems. This program can achieve the processing speed that is almost proportional to the number of cores.

I. 서론

멀티 코어 시스템은 많은 디지털 신호처리 알고리즘의 빠른 처리를 가능하게 해준다. 그러나 feedback이 필요한 recursive filtering이나 adaptive filtering 에서는 dependency problem 때문에 병렬 처리를 통하여 수행속도를 증가시키는 것이 어렵다. 예를 들어, 1차의 recursive filtering 식인 $y[n]=ay[n-1]+x[n]$의 경우 한 번에 여러 개의 출력을 구하는 것이 직접 계산을 통해서는 불가능하다. 본 논문에서는 최근에 널리 보급되는 공유메모리 멀티코어 환경에서 recursive filtering을 OpenMP (Open Multi-Processing) API를 이용하여 구현하였다.

이 논문은 지식경제부 출연금으로 ETRI와 시스템반도체산업진흥센터에서 수행한 ITSoC 핵심설계인력양성사업과 교육과학기술부의 재원으로 한국학술진흥재단에서 수행하는 BK21 프로젝트의 지원을 받아 수행된 연구입니다.

II. Recursive Filtering의 병렬계산

Recursive digital filtering에 사용되는 M차의 식은 다음과 같다.

$$y[n]=\sum_{i=1}^{M}a_i y[n-i]+\sum_{j=0}^{M-1}b_j u[n-j] \quad \text{식(1)}$$

위의 식의 처리를 위해 입력은 P개의 블럭으로 나누어지며, 매 블럭 당 샘플의 개수는 L로 가정 한다. 즉 전체 입력 데이터의 개수는 PL이다. 식(1)의 해는 particular solution과 transient solution으로 구성되어있다. 이 중 particular solution은 input data인 $u[n]$만으으로 계산이 가능하다. 즉 particular solution을 구하는 데에는 각 블럭의 초기조건이 불필요하므로, 블럭 당 독립적 처리가 가능하다. 반면에 transient solution은 초기 조건에만 의존하는 해로써 초기 조건에 적당한 계수를 곱하여 구한다 [1].

본 연구에서 사용된 방법에서는 우선, 입력을 여러 개의 블록으로 나누어 $u[i,j]$로 만든다. 이 때 i는 블럭의 index, j는 그 블럭 내의 index값이다. 알고리즘의 첫 단계(step 1)에서는 모든 블록(첫 번째 블럭 제외)의 초기조건을 0으로 하여 particular solution, $z[i,j]$,을 구한다. 각 블럭 내의 particular solution은 다음과 같이 구해진다.

$$z[i,j]=\sum_{k=1}^{M}a_k z[i,j-k]+\sum_{k=0}^{M}b_k u[i,j-k] \quad \text{식(2)}$$

위의 식을 계산 시, 각 블럭의 초기 조건이 0이므로 각 블록간(index i)에는 독립적이며 각 블럭을 하나의

CPU-core (즉 하나의 thread)로 처리한다. 본 계산은 복수의 thread를 이용 빠르게 처리될 수 있으나, 각 블럭의 초기조건을 0으로 놓고 계산했기 때문에 불완전한 답이다. 총 계산 시간은 $2LM/P$로 모델된다.

다음으로, $z[i,j]$에 각 블록의 초기조건에 적당한 계수를 곱하여 식(3)과 같이 transient solution을 구하고 그것을 $z[i,j]$에 더하여 $y[i,j]$를 구한다. 이 때 블록 내에서는 병렬처리를 할 수 있다. 그러나 첫 번째 블럭을 제외하고서는 초기조건이 없으므로 이것을 미리 계산해야 한다. 참고로 CA^{j+1}은 미리 계산해둔다.

$$y(i,j) = CA^{j+1}\overline{y(i,-1)} + z(i,j) \qquad \text{식(3)}$$

$$,C = [1\ 0\ 0\ \cdots\ 0],\quad A = \begin{bmatrix} a_1 & a_2 & \cdots & a_M \\ 1 & 0 & \cdots & 0\ \ 0 \\ 0 & 1 & \cdots & 0\ \ 0 \\ 0 & 0 & \cdots & 1\ \ 0 \end{bmatrix}$$

식 (3)을 이용하면, 첫 번째 블럭의 초기조건에서 M^2번의 계산으로 두 번째 블럭의 초기조건을 구할 수 있다. 그 까닭은 식(3)의 계산은 j에 독립적이기 때문에 중간의 계산을 생략하고 맨 마지막 샘플들의 transient solution을 구하는 것이 가능하기 때문이다. 이 두 번째 블럭의 초기조건을 이용하여 다시 M^2번의 계산으로 세 번째 블럭의 초기조건을 구할 수 있다. 이 때 이 과정은 병렬화가 안 된 채로 실행이 되지만, 각 블럭의 길이(L)가 충분히 길다면, 블럭 내의 중간 값을 계산하지 않기 때문에 매우 빠르게 진행된다 (look-ahead step). 따라서 총 P개의 블럭을 처리 할 때, 걸리는 시간은 PM^2로 모델할 수 있다. 이상이 두 번째 과정(Step 2)이다.

세 번째 단계에서는 각 블럭마다 독립적으로 transient solution을 모두 계산하는 과정이다. 이는 (3)을 이용하며 각 블럭마다 모두 독립적으로 실행할 수 있으므로, 총 계산 시간은 LM/P로 모델할 수 있다.

Ⅲ. 구현

OpenMP로 구현한 코드는 다음과 같다.

```
#pragma omp parallel for private(j,k)
for i=0 to P-1
  for j=0 to L-1
  z(i,j) = Σ_{k=1}^{M} a_k z(i,j-k) + Σ_{k=0}^{M} b_k u(i,j-k)

for i=0 to P-1
  for j=L-M to L-1
      y(i,j) = CA^{j+1} y(i,-1)) + z(i,j)
      if (i<P-1) y(i+1,j-L) = y(i,j)

#pragma omp parallel for private(j)
for i=0 to P-1
  for j=0 to L-1   y(i,j) = CA^{j+1} y(i,-1)) + z(i,j)
```

위의 코드에서 Step 1의 for loop에서는 각 블록이 독립적이므로 바깥쪽 for loop을 병렬 처리 할 수 있다. Step 2 에서는 각 블록의 초기조건이 순차적으로 계산되며 단 하나의 CPU-core만이 사용된다. 그리고 Step3에서도 Step 1과 마찬가지로 각 블럭을 하나의 CPU-core가 맡아서 병렬처리 한다.

만약 LP개의 샘플을 P개의 블럭으로 나누어서 동시에 처리하는 경우를 가정하면, 직렬알고리즘으로는 $2MLP$개의 계산이 필요하고, 한 샘플 당 계산량은 $2M$번이다. 병렬처리 방법의 경우 전체 계산량은 $3LM/P + PM^2$이며, 샘플 하나당 계산량은 $3M/P + M^2/L$이 되어서, 블럭 사이즈 L이 크다면 직렬 알고리즘에 비해서 약 50% 정도의 오버헤드만으로 병렬처리가 가능하다.

Ⅲ. 실험 결과 및 결론

Intel core2 Quad 2.4GHz의 Cpu 환경에서 100M개의 입력 데이터를 사용하였다. 프로세서(P)를 2개, 4개 사용하여 위의 알고리즘을 수행하였다.

Time sec. (Speedup %)	M=2	M=4	M=8	M=16
Sequential	0.690(100)	1.061(100)	1.480(100)	3.022(100)
P=2	0.582(119)	0.819(129)	1.121(132)	2.363(128)
P=4	0.577(120)	0.605(175)	0.602(246)	1.202(251)

[표 1] 코어 수(P)와 필터차수(M)에 따른 수행시간 비교

연산량이 적은 M=2, 4 에서는 충분한 speed-up을 얻지 못하였으나, 연산량이 큰 M=8, 16에서는 4개의 CPU-core를 사용할 때 240%가 넘는 속도향상을 얻을 수 있었다.

참고문헌

[1] W. Sung and S. K. Mitra, “Efficient Multi-Processor Implementation of Recursive Digital Filters”, *IEEE ICASSP* 1986 Tokyo.

[2] W. Sung, S. K. Mitra, and B. Jeren, "Multiprocessor Implementation of Digital Filtering Algorithm Using a Parallel Block Processing Method", *IEEE Tran. on Parallel and Distributed Systems,* vol. 3, no. 1 Jan 1992